High Level Parallel Compositions (CPANs) for the Parallel Programming based on the use of Communication Patterns

Mario Rossainz López[1], Manuel I. Capel Tuñón[2]

[1] Benemérita Universidad Autónoma de Puebla, Avenida San Claudio y 14 Sur, San Manuel, Puebla, State of Puebla, 72000, México
mariorl@siu.buap.mx
http://www.cs.buap.mx/~mrossainz
[2] Departamento de Lenguajes y Sistemas Informáticos, ETS Ingeniería Informática, Universidad de Granada, Periodista Daniel Saucedo Aranda s/n, 18071, Granada, Spain
mcapel@ugr.es
http://lsi.ugr.es/~mcapel

Abstract. This article presents a programming methodology based on High Level Parallel Compositions (CPAN in the Spanish acronym) within a methodological infrastructure made up of an environment of Parallel Objects [10], an approach to Structured Parallel Programming and the Object-Orientation paradigm. The implementation of commonly used communication patterns is explained by applying the method (the CpanFarm, CpanPipe and CpanTreeDV that represent respectively, the patterns of communication Farm, Pipeline and Binary Tree, the latter one used within a parallel version of the design technique known as Divide & Conquer), which conforms a library of classes suitable for use in applications within the programming environment of the C++ and POSIX standards for thread programming. Thus, in this work presents the design of the CPAN that implements a parallelization of the algorithmic design technique named Branch & Bound and uses it to solve the Travelling Salesman Problem (TSP).

1 Introduction

Obtaining efficiency in parallel programs is not so much a problem of acquiring processor speed, but rather, it is about how to program efficient interaction/communication patterns among the processes [1], [2], [4], [6] to achieve the maximum possible speed-up of a given parallel application. Parallel Programming based on the use of communication patterns is known as Structured Parallel Programming (SPP) [6], [7]. The widespread adoption of SPP methods by programmers and system analysts currently presents a series of open problems. We are particularly interested in proposing new solutions to the following: (a) the lack of SPP methods applicable to the development of a wider range of software applications; (b) the determination of a complete set of communication patterns and their semantics; (c) the

© *A. Gelbukh, S. Torres, I. López (Eds.)*
Advances in Computer Science and Engineering
Research in Computing Science 19, 2006, pp. 29-40

necessity to make predefined communication patterns or high level parallel compositions available to the community, aimed at encapsulating parallel code within programs; (d) the adoption of a sound (i.e. without *anomalies*) programming approach based on merging concurrent primitives and Object-Oriented (O-O) features, thereby meeting the requirements of *uniformity, genericity* and *reusability* of software components [6]. The present investigation is focused on SPP methods, and a new implementation is proposed (carried out with C++ and the POSIX Threads Library) of a library of High Level Parallel Composition (CPAN) [6], [7] classes, which provide the programmer with the communication patterns most commonly used in Parallel Programming. At the moment, the library includes the following ones: CpanFarm, CpanPipe, CpnaTreeDV, the latter one being used in a parallel version of Divide & Conquer algorithmic design technique and CpanFarmBB that is one pattern composed with Farm process that implements a parallelization of the algorithmic design technique named Branch & Bound.

1.1 The Problem Being Tackled

In order to cope with the above described items, we have found that an O-O Parallel Programming environment providing the features listed below must be used, (a) capacity of object method invocation that assumes asynchronous message passing and asynchronous futures; (b) the objects should have internal parallelism; (c) availability of different communication mechanisms when service of petitions from client processes take place in parallel; (d) distribution transparency of processes within parallel applications; (e) Programmability, portability and performance, as a consequence of software development within an O-O programming system.

1.2 Scientific Objectives in this Research

The current investigation has mostly been carried out within the PhD thesis research work referenced in [8], whose achieved operational objectives are listed below:

1. To develop a programming method based on High Level Parallel Compositions or CPANs.
2. To develop a library of classes of parallel objects [10] that provides the programmer or the analyst with a set of commonly used communication patterns for parallel programming; the objects should be uniformly programmed as reusable, generic, CPANs.

To offer this library to the programmer, so that he/she can exploit it by defining new patterns, adapted to the communication structure of processes in his/her parallel applications, by following an O-O programming paradigm, which includes class inheritance and object generic instantiation as its main reusability mechanisms.

2 High Level Parallel Compositions or CPANs

The basic idea of the programming method consists of the implementation of any type of communication patterns between parallel processes of an application or distributed/parallel algorithm as CPAN classes, following the O-O paradigm. CPANs are aimed at helping parallel applications programmers in programming efficient, portable and easy to program code by encapsulating parallelism or communication protocols from the sequential application processes of the parallel applications [8]. CPANs are structured as three classes of parallel objects [10], see Fig 1:

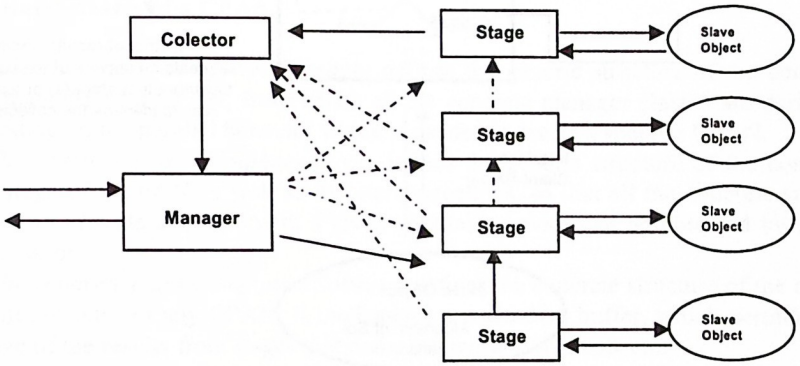

Fig. 1. Internal Structure of a CPAN

An object manager, which is the only visible interface to the sequential processes in a parallel application, composed of the collector and stages objects and should be coordinated by the manager itself, see Fig 2.

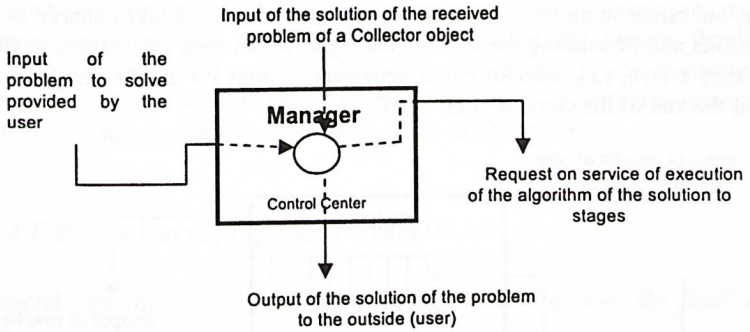

Fig. 2. The Manager Object (Internal Structure)

The stage objects intended to configure a connection topology among these objects in order to provide a given communication pattern semantics. The stage objects

are objects of specific purpose responsible for encapsulating a client-server type interface between the manager and the object slaves (objects that are not actively participative in the composition of the CPAN, but rather, are considered external entities that contain the sequential algorithm constituting the solution of a given problem), see Fig 3.

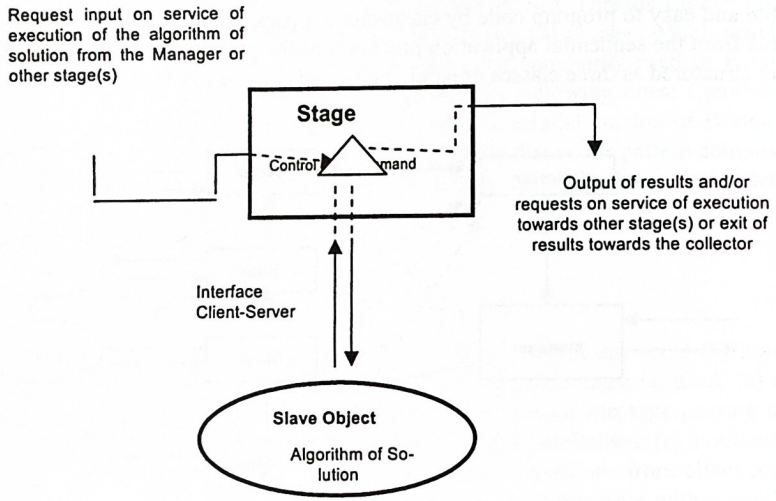

Fig. 3. The Stage Object (Internal Structure)

An object collector in charge of storing in parallel the results received from the stages during the service of a sequential process petition. The control flow within the stages of a CPAN depends on the communication pattern implemented between these. When the CPAN concludes its execution, the result does not return to the manager directly, but rather to an instance of the class Collector, which takes charge of storing these results and of sending them to the manager, which then sends them to the exterior as they arrive, i.e., without begin necessary to wait for all the results to be obtained at the end of the computation. See Fig 4.

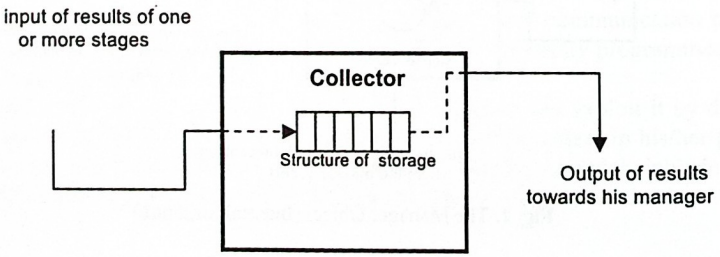

Fig. 4. The Collector Object (Internal Structure)

2.1 Types of Communication Between the Parallel Objects

1. *The synchronous way* stops the client's activity until the object's active server gives back the answer to the petition.
2. *The asynchronous way* does not force any waiting in the client's activity; the client simply sends its petition to the active server and then it continues.
3. *The asynchronous future way* makes only to wait the client's activity when the result of the invoked method is needed to evaluate an expression during its code execution.

2.2 Basic classes of a CPAN

The abstract class ComponentManager defines the generic structure of the component manager of a CPAN, from which all the concrete manager classes are derived, depending on the parallel behavior which is needed to create a specific CPAN.

The abstract class ComponentStage defines the generic structure of the component stage of a CPAN as well as its interconnections, so that all the concrete stages needed to provide a CPAN with a given parallel behavior can be obtained by class instantiation.

The concrete class ComponentCollector defines the concrete structure of the component collector of any CPAN. It implements a multi-item buffer, which permits the storage of the results from stages that make reference to this collector.

2.3 The Synchronization Restrictions MaxPar, Mutex and Sync

Synchronization mechanisms are needed when several petitions of service take place in parallel in a CPAN, being capable its constituting parallel objects of interleaving their concurrent executions while, and at the same time, they preserve the consistency of the data being processed [10]. Within the code of any CPAN, execution constraints are automatically included when the methods MaxPar (Maximum Parallelism), MutEx (Mutual Exclusion) and Sync (Synchronization type producer-consumer) of the library are called. The latter ones must be used to obtain a correct programming of object methods and to guarantee data consistency in applications.

3 The CPANs Farm, Pipe and TreeDV

The parallel patterns applied until now have been the *Pipeline*, the *farm* and the *treeDV*.

The Pipeline is made up of a set of interconnected stages, one after another, in which the information flows between these until an ending condition is determined in one of them. At this moment the pipeline enters in another execution mode in which each stage unloads its data to the next one. The last stage is responsible for sending the processes data to the Collector. See Fig 5.

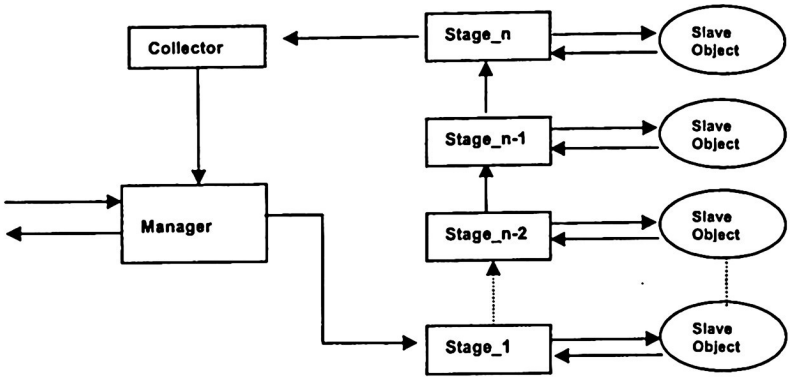

Fig. 5. The CPAN of a Pipeline

The *Farm* is composed of a set of worker processes executed in parallel until a common objective is reached, and a controller in charge of distributing work and controlling the progress of the global calculation. See Fig 6.

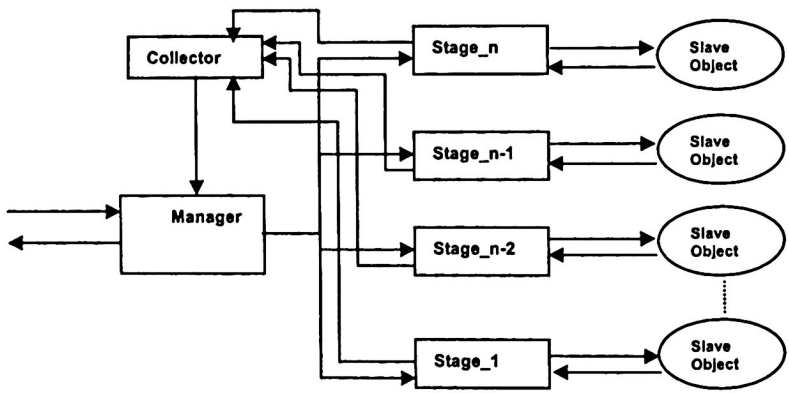

Fig. 6. The CPAN of a Farm

The *TreeDV* is a communication pattern in which the information flows from the root to the leaves of the tree and vice versa. The nodes on the same level are executed in parallel in order to implement a parallel version of the so called Divide & Conquer algorithmic design technique. The stage situated at the root of the TreeDV will obtain the solution of the problem when the global calculation finishes. This CPAN is configured in a similar way. See Fig 7.

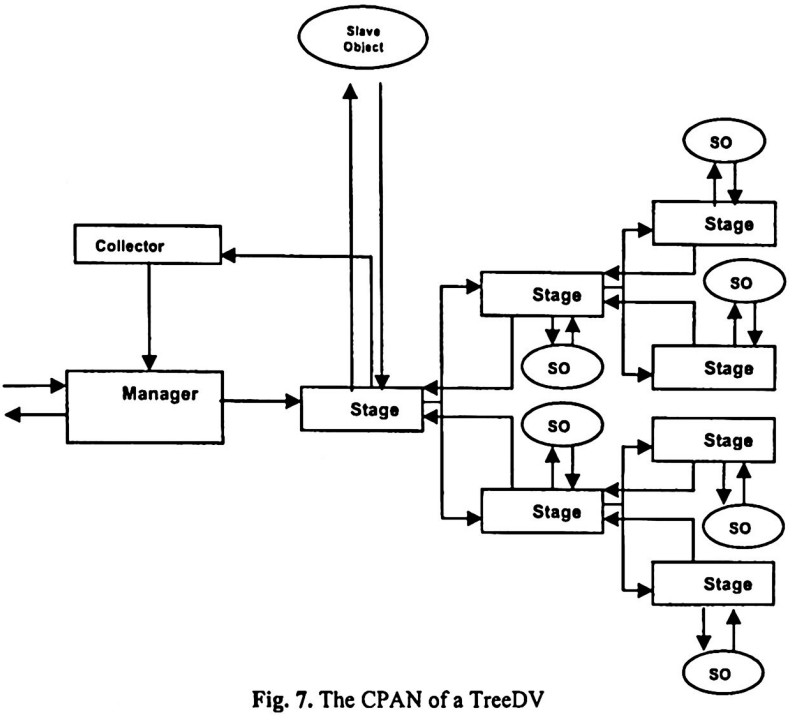

Fig. 7. The CPAN of a TreeDV

These constitute a significant set of reusable communication patterns in multiple parallel applications and algorithms. See [5] and [8] for details.

3.1 Results Obtained

Some CPANs adapt better to the communication structure of a given algorithm than others, therefore yielding different speedups of the whole parallel application. The way in which it must be used to build a complete parallel application is detailed below.

1. It is necessary to create an instance of the adequate class manager, that is to say, a specialized instance (this involves the use of inheritance and generic instantiation) implementing the required parallel behavior of the final manager object. This is performed by following the steps:
   1.1. Instance initialization from the class manager, including the information, given as associations of pairs (slave_obj, associated_method); the first element is a reference to the slave object being controlled by each stage and the second one is the name of its callable method.
   1.2. The internal stages are created (by using the operation *init()*) and, for each one, the association (slave_obj, associated_method) is passed to. The second element is needed to invoke the associated_method on the slave object.

2. The user asks the manager to start a calculation by invoking the *execution()* method of a given CPAN. This execution is carried out as it follows:
   2.1. a collector object is created for satisfying this petition;
   2.2. input data are passed to the stages (without any verification of types) and a reference to the collector;
   2.3. results are obtained from the object collector;
   2.4. The collector returns the results to the exterior without type verification.
3. An object manager will have been created and initialized and some execution petitions can then start to be dispatched in parallel.

We carried out a Speedup analysis of the Farm, Pipe and TreeDV CPANs for several algorithms in an Origin 2000 Silicon Graphics Parallel System (with 64 processors) located at the European Center for Parallelism in Barcelona (Spain), this analysis is discussed below.

Assuming that we want to sort an array of data, some CPANs will adapt better to communication structure of a Quicksort algorithm than others. These different parallel implementations of the same sequential algorithm will therefore yield different speedups. The program is structured of six set of classes instantiated from the CPANs in the library High Level Parallel Compositions, which constitute the implementation of the parallel patterns named Farm, Pipe and TreeDV. The sets of classes are listed below:

1. *The set of the classes base*, necessary to build a given CPAN.
2. The set of the classes that define the abstract data types needed in the sorting.
3. *The set of classes that define the slave objects*, which will be generically instantiated before being used by the CPANs.
4. The set of classes that define the Cpan Farm.
5. The set of classes that define the Cpan Pipe.
6. The set of classes that define the Cpan TreeDV.

This analysis of speedup of the CPANs appears in Figures 8, 9 and 10. In all cases the implementation and test of the CPANs Farm, Pipe and TreeDV 50000 integer numbers were randomly generated to load each CPAN.

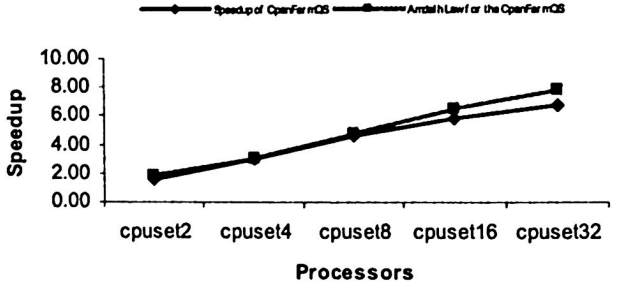

Fig. 8. Scalability of the Speedup found for the CpanFarm in 2, 4, 8, 16 and 32 processors

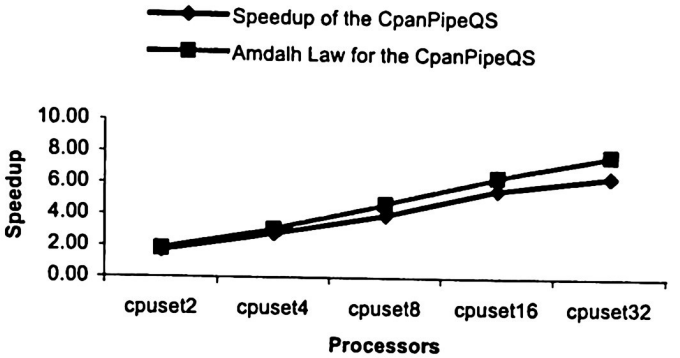

Fig. 9. Scalability of the Speedup found for the CpanPipe in 2, 4, 8, 16 and 32 processors

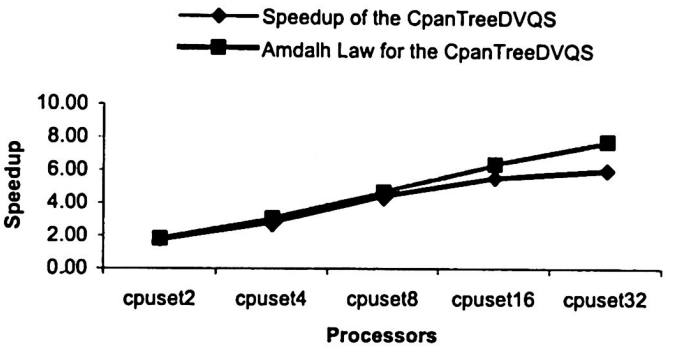

Fig. 10. Scalability of the Speedup found for the CpanTreeDV in 2,4,8, 16 and 32 processors

4 Parallelization of the Branch & Bound Technique

Branch-and-bound (BB) makes a partition of the solution space of a given optimization problem. The entire space is represented by the corresponding BB *expansion tree*, whose root is associated to the initially unsolved problem. The children nodes at each node represent the subspaces obtained by *branching*, i.e. subdividing, the solution space represented by the parent node. The leaves of the BB tree represent nodes that cannot be subdivided any further, thus providing a final value of the cost function associated to a possible solution of the problem.

Three stages are performed during the execution of a program based on a BB algorithm:

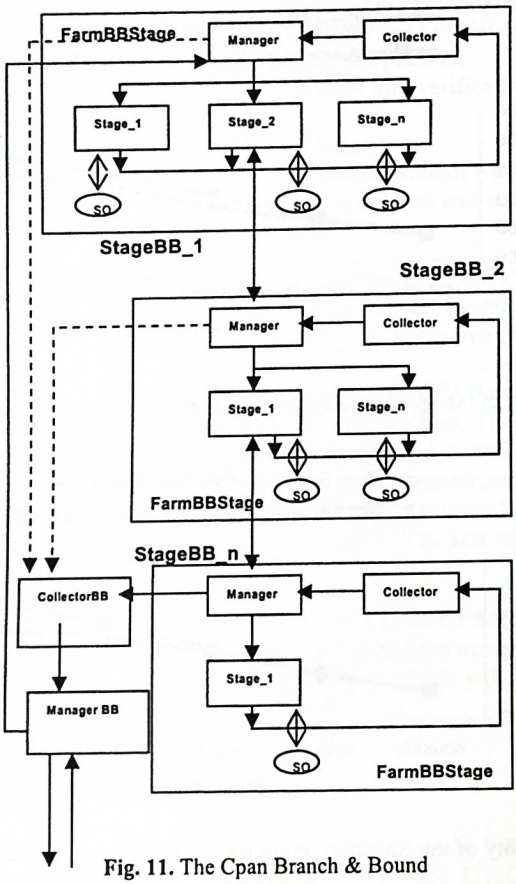

Fig. 11. The Cpan Branch & Bound

1. *Branch:* the node selected in the previous step is subdivided in its children nodes by following a ramification scheme to form the expansion tree. Each child receives from its father node enough information to enable it to search a suboptimal solution.
2. *Bound:* Some of the nodes created in the previous stage are deleted, i.e. those whose partial cost, which is given by the cost function associated to this BB algorithm instance, is greater than the best minimum bound calculated up to that point.

The ramification is generally separated from the bounding of nodes on the expansion tree in parallel BB implementations, and so we followed this approach using *a Farm communication scheme* [9]. The expansion tree, for a given instance of the BB algorithm, is obtained by iteratively subdividing the stage objects according to this pattern until a stage representing a leaf-node of the expansion tree is found, see Fig 11.

The pruning is implicitly carried out within another *farm* construction by using a *totally connected scheme* between all the processes. The manager can therefore communicate a sub-optimal bound found by a process to the rest of the branching processes and thus avoid unnecessary ramifications of sub-problems. *The Cpan Branch & Bound* is composed of a set of *Cpans Farm*; see Figure 11, which represent each one a set of worker processes and one manager, therefore, forming a new type of structured Farm, the *Farm Branch & Bound or FarmBB*, which is also included in the library of CPANs. All the worker processes of the *Farm BB* are executed in parallel, thereby forming the expansion tree of nodes given by the BB algorithm technique. The initial problem, or the root of the expansion tree, is given to the manager process of the initial *Cpan Farm*, which is in charge of distributing the work and of controlling the global calculation progress. It is also responsible for sending results to the collector of the *Cpan FarmBB*, which will display them [9].

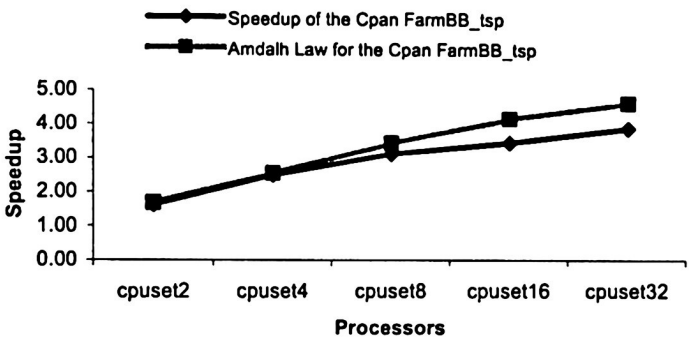

Fig. 12. Speedup of parallel CpanBB with N=50 cities in 2, 4, 8, 16 and 32 processors

The CPAN based parallel BB algorithm was tested by solving the TSP with 50 cities and by using the first best search strategy driven by a *least cost* function associated to each live node. The results obtained yielded a deviation ranging from 2% (2 processors) to 16% (32 processors) with respect to the optimal ones, as predicted by the Amdalh law for this parallelized algorithm. See Fig 12.

5 Conclusions

The programming method presented is based on Corradi's High Level Parallel Compositions, but updated and adapted to be used with the C++ programming language and POSIX standard for thread programming. The CPANs Pipe, Farm, and TreeDV comprise the first version of a library of classes intended to be applied to solve complex problems such as the afore-mentioned parallelization of the Branch & Bound technique, thus offering an optimal solution to the TSP NP-Complete problem.

References

1. Brinch Hansen; "Model Programs for Computational Science: A programming methodology for multicomputers", Concurrency: Practice and Experience, Volume 5, Number 5, 407-423, 1993.
2. Brinch Hansen; "SuperPascal- a publication language for parallel scientific computing", Concurrency: Practice and Experience, Volume 6, Number 5, 461-483, 1994.
3. Capel M.I., Palma A., "A Programming tool for Distributed Implementation of Branch-and-Bound Algorithms". Parallel Computing and Transputer Applications. IOS Press/CIMNE. Barcelona 1992.
4. Capel, M.; Troya J. M. "An Object-Based Tool and Methodological Approach for Distributed Programming". Software Concepts and Tools, 15, pp. 177-195. 1994.
5. Capel, M.; Rossainz, M. "A parallel programming methodology based on high level parallel compositions". Proceedings of the 14th International Conference on Electronics, Communications and Computers, 2004, IEEE CS press. 0-7695-2074-X.
6. Corradi A, Leonardo L, Zambonelli F. "Experiences toward an Object-Oriented Approach to Structured Parallel Programming". DEIS technical report no. DEIS-LIA-95-007. 1995
7. Danelutto, M.; Orlando, S; et al. "Parallel Programming Models Based on Restricted Computation Structure Approach". Technical Report-Dpt. Informatica. Universitá de Pisa.
8. Rossainz, M. "Una Metodología de Programación Basada en Composiciones Paralelas de Alto Nivel (CPANs)", Universidad de Granada, PhD dissertation, 02/25/2005.
9. Rossainz M, Capel M. "Design and use of the CPAN Branch & Bound for the solution of the traveling salesman problem (TSP)". Proceedings of the ECMS 2005 – HPC&S. Riga Latvia, 2005. ISBN: 1-84233-113-2.
10. Rossainz M, Capel M. "An Approach to Structured Parallel Programming Based on a Composition of Parallel Objects". Congreso Español de Informática CEDI-2005. XVI Jornadas de Paralelismo. Granada, Spain 2005. Editorial Thomson. ISBN: 84-9732-430-7.